Nike's Software Architecture and Infrastructure: Enabling Integrated Solutions for Gigahertz Designs

V. Nagbhushan, Nike Development, DT, Intel Corp.
Yehuda Shiran, Nike Development, DT, Intel Corp.
Satish Venkatesan, Nike Development, DT, Intel Corp.
Tamar Yehoshua, Nike Development, DT, Intel Corp.

Index words: architecture, data model, software infrastructure

Abstract

This paper describes how Nike's innovative architecture addresses the expanding requirements of Intel's next-generation processor designs while enabling a design environment that is more productive than one built with the previous tool generation.

This paper shows how software architecture and data modeling techniques are used as core attributes of a CAD tool suite. We discuss the issues that have influenced Nike's architectural direction, such as technology trends, processor architecture trends, and computing platform trends. We identify some of the major drawbacks of existing tool suites and show how Nike architecture addresses these. Lastly, we describe how we developed a software infrastructure in order to support and facilitate the code sharing necessary to implement the designed architecture. The standard software development environment is described, including the tools and methodologies that are uniformly deployed to all Nike developers.

Introduction

Design and Test Technology (DT) is the supplier of CAD/CAE solutions for Intel's lead processor design projects. The Nike department within DT is chartered to provide the future generation CAD tool suite for use well into the next millennium. The first release of the Nike tool suite is scheduled for Q3, 1999.

Nike architecture was influenced by several external and internal vectors. The primary external vectors were industry technology trends, processor design trends, and Intel's design roadmap and computing platform trends. The primary internal vectors were the need to improve development efficiency, tool quality and maintainability, and to ensure adequate extendibility.

External Vectors

Industry technology trends predict a continuation of feature size reduction resulting in an exponential increase in chip transistor counts and a significant increase in frequency. Chips in the next decade could have upwards of 100 million transistors and run at frequencies well beyond 10GHz. As feature sizes decrease, aggressive circuit styles are also becoming the norm. This implies that second order effects, such as noise and inductance, become key factors in design decisions. In order to make these decisions effectively and efficiently, designers need increased visibility into data from multiple domains, such as circuit and layout.

Analysis of microprocessor architecture trends [1] and the Intel design roadmap show the emergence of highly integrated chips and ever decreasing time to market. This combination necessitates a design environment that will enable significant improvements in designer productivity.

A major change in the computing environment has been the emergence of Windows NT * on Intel® architecture (IA) as a sophisticated, inexpensive, and powerful platform. In comparison to the existing UNIX* environment commonly used for CAD, the Windows* environment provides a more consistent user interface (UI) as well as a rich set of office applications. The challenge for the developers of new CAD tool suites is to integrate the office environment with the CAD tools and use the Windows environment to provide a more productive system for the user.

* All other trademarks are the property of their respective owners.

Internal Vectors

As the coverage of the tool suite and the overall software size increases, improving software efficiency becomes very important. Software efficiency can be broadly categorized along two lines:

- *Extensibility and maintainability* of CAD tools. Even though individual tools might change, the core architecture of a tool suite persists for a very long time. For example, the previous CAD architecture at Intel spanned a decade. Given the dynamic nature of VLSI technology, it is impossible to predict accurately for the next decade. Hence, Nike architecture should be easily extendible and efficiently maintainable. For example, we need to build in enough headroom so that new manufacturing process features can be incorporated without massive system-wide code changes. Similarly, tools must be customizable to allow major changes in design methodology.
- *Tool quality and development efficiency.* In order to reduce the number of iterations in the design and implementation of complex software, there needs to be a well defined software development methodology where developers have detailed specifications and implementation plans upfront. In addition, quality needs to be built into the tools in order to avoid numerous cycles to fix defects.

In the next section we describe the Nike software architecture, goals, and principles. We then present a data-modeling methodology and architecture that are key enablers for achieving Nike architectural objectives. In the subsequent section, we describe a software infrastructure that supports and facilitates efficient development of Nike CAD tools.

Nike Software Architecture

This section describes a novel software architecture pioneered by Nike. We examine the drawbacks of existing CAD architectures. We then present LaMA, a layered modular architecture, followed by a set of architectural principles that drive the design and development of Nike.

Drawbacks of Existing Architectures

After studying existing CAD architectures, we found that they have several deficiencies. The following is a brief overview of the salient root causes of these deficiencies and the symptoms that they exhibit:

- *Non-modular*. Software was not written in a modular fashion. This often led to local implementations of similar or even identical functionality, resulting in inconsistent behavior. Users had to reconcile inconsistencies such as different timing or RC modules in various tools. It also made reuse difficult and severely impacted development efficiency. One of the causes of non-modularity was the existence of several data models; for example, multiple data representations for layout.
- *Difficult data exchange across domains*. Multiple data models in various domains were inconsistent in terminology, interfaces, and implementation. This made it difficult to provide a unified mechanism to map entities across the domains. Such mapping, when necessary, was done in an ad-hoc fashion that often resulted in loss of data, and consequently, productivity.
- *Ad hoc persistence mechanisms*. Typically, ASCII files were used to store data persistently. However, there was inadequate use of industry standard formats, often leading to a proliferation of files, multiple readers/writers, and semantic mismatches between formats representing the same data. ASCII files are also not performance oriented, have fundamental capacity limitations, are intolerant of new software releases, and make it very difficult to implement incremental input/output.
- *Inconsistent look and feel*. Users need to familiarize themselves with different interfaces to perform similar actions. This results in a high learning curve as well as loss of productivity.

Layered Modular Architecture (LaMA)

Nike is pioneering Layered Modular Architecture (LaMA), a hierarchical decomposition of a complex tool suite consisting of over a hundred tools with a total code size exceeding a million lines of code. Figure 1 shows the basic LaMA pyramid.

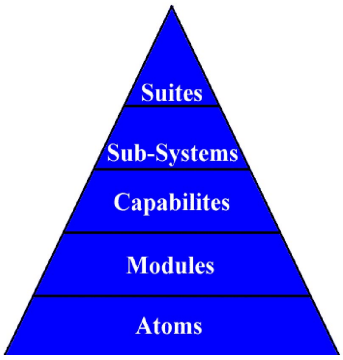

Figure 1: Layered modular architecture

In this model, the overall tool suite consists of several sub-systems; each sub-system is targeted at a particular user flow. A sub-system is comprised of sev-

eral capabilities each of which represents a user-visible functionality. A capability is implemented by one or more software modules, which are in turn made up of several atoms. The modules and atoms are designed with well-defined interfaces to facilitate reuse by several capabilities.

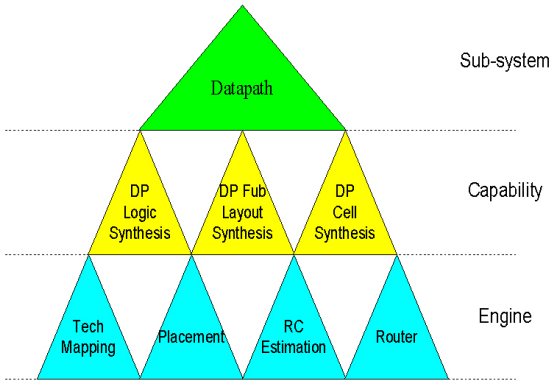

Figure 2: Example of LaMA sub-system

Figure 2 shows an example sub-system and its decomposition. Users may interact with these sub-systems through a visual cockpit, which enables simultaneous interaction with multiple domains such as circuit and layout. This reduces the number of design iterations caused by downstream surprises. The modular architecture enables a software developer to reuse the modules and atoms, and the cockpit integrator to reuse the capabilities to the maximum extent.

Software Architecture Principles

The ultimate objective of Nike software architecture is to enable efficient development of the CAD software that can meet or beat the high-level specifications set forth for the system. In order to drive this objective, we formulated a set of architectural principles. They were kept general enough so as to apply to all areas of the Nike tool suite; functional and area-dependent factors (e.g., principles that may apply only to timing tools) are not covered here. These main architectural principles are briefly explained below:

- Provide *integrated tool suites (or cockpits)* that enable users to execute a flow or perform similar operations. The primary goal is to improve user productivity.
- Software components should have a *modular design* and should be implemented *using common, unified services*. Each component should have a well documented interface, and similar functional components across the entire Nike tool suite should be implemented as common services. For example, there should only be one RC estimator for a given input abstraction, accuracy, and runtime. This can significantly improve reuse, quality, development efficiency and end-user consistency.
- The Nike system should present a *common look and feel*. The look (visual appearance) of UI objects that perform similar operations should be similar. The feel (behavior) of similar operations and data objects should be similar. The goal is to improve the predictability of the system, and consequently, the productivity of the user.
- Tools should support *incremental processing*, that is, they should be able to handle small delta changes to the input by incremental processing, and produce results that exhibit small delta changes.
- The Nike system should be *extensible.* Both, individual capabilities, as well as the entire tool suite should allow a user to easily extend its functionality.
- The Nike system should support *plug and play* with external tools.

A subsequent section describes a data modeling methodology and architecture that embodies these principles and enables us to achieve the architectural objectives. While a detailed description of the entire Nike architecture is beyond the scope of this paper, the following section describes the Nike layout architecture and how it meets the goals stated above.

Nike Layout Software Architecture

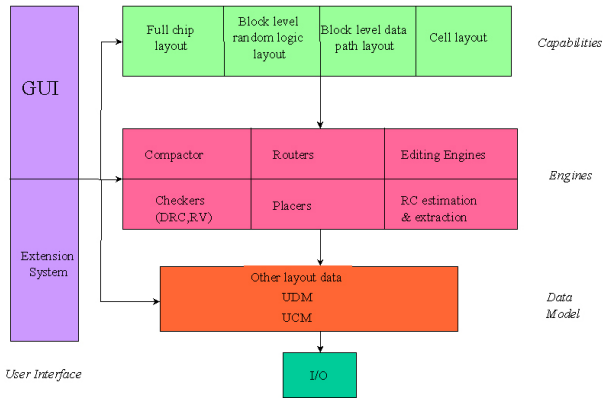

Figure 3: Nike layout architecture

Figure 3 shows a layered view of the Nike layout software architecture where most of the software modules have been partitioned into four layers: data model, engines, capabilities, and UI. A software module can, in general, access any module in a layer below it, either directly or indirectly (e.g., the full chip module

can make calls to any of the engines or data modules). Each module has a well defined procedural interface (API), which can be accessed by other modules.

The data model (DM) layer is the foundation of the architecture and consists of the software that models CAD data in memory and its interfaces. All the modules access CAD data by making calls to the DM APIs. The engines layer comprises mostly algorithmic modules (such as design rule checkers) that perform analysis and synthesis operations on the data. Modules in the capabilities layer address requirements of a subset of the domain (e.g., full-chip module). The user interface (UI) layer, drawn vertically, represents the user interface presented by all the other modules. The I/O module below the DM layer is a specialized engine to provide file input/output and persistence services. It accesses a slightly lower-level DM interface (compared to other engines) to enable fast I/O. The following paragraphs provide details about each layer:

- The *data model* layer serves as the in-memory repository for all the primary, non-derivable data in the system. In the case of layout, it is called the unified data model (UDM) and contains data (such as cells, wires, nets, transistors, etc.) and functions to access and modify the data. The data is modeled as a hierarchical class system, which is described in the next section. The API to this layer guarantees consistency of the data structures and semantics.
- The *engines* layer is comprised of software modules with well defined functionality. Algorithmic engines (such as RC estimation and extraction, placement, global and detailed routing (GR, DR), compaction, netlisters) and core editing engines (wire editing, move, etc.) fall into this layer. All the engines work off data from the DM layer, but may create temporary, derived data for efficiency. For example, the DRC engine works off scan-line data that is derived from the DM data. The derived application data may be saved along with the primary UDM data by the persistence mechanism to enable incremental processing.
- The capabilities layer is mainly comprised of environments that address the traditional sub-domains such as full-chip layout, block layout (for random logic and datapath) and leaf cell. Each of these provides functionality and customization appropriate to that capability. Each capability can either directly expose an engine functionality to the user (through the UI layer) or hide or modify it.
- The user interface (UI) layer provides user visibility into functionality and data. It is comprised mainly of a graphical user interface (GUI) and extension system. The capabilities and some engines instantiate GUI items at run-time to interact with the user. Engines such as DRC provide their own GUI to customize rules and display and scan errors. This enables them to be completely self-contained and reusable. The extension system enables customization by allowing easy access to data and functionality. The data model, engine and capability modules, provide access through a Tcl interpreter and through Windows* automation interfaces. GUI customization is enabled by VisualBasic* for Applications (VBA) on Windows NT*.

All the capabilities shown in Figure 3 are provided to the user in an integrated framework called the Nike Integrated Layout Environment (NILE).

The layered architecture ensures that there are no loops in the module dependency graph. Each module has a well defined API to maximize reusability; for example, the engines are reused by several capabilities. This re-usability has enabled us to develop new capabilities very quickly.

We have invested significant effort to ensure that each function is implemented by only one software module. For example, there is only one RC estimation module. If implementation of multiple modules began simultaneously, they were merged after the requirements matured.

The common GUI layer guaranteed a common look (e.g., native NT-looking widgets, docking windows, common list, and tree viewers, etc.). The I/O module is the single entry and exit channel for all design data. This has enabled us to minimize semantic mismatches.

Data Modeling

From a data modeling perspective, the processor design cycle can be divided into three phases: logic, circuit, and layout. Each phase successively refines the level of abstraction by adding details.

The design cycle involves many iterations, both within a phase as well as between phases. At each phase, a new representation is created, analyzed, and iteratively improved to meet system requirements. One objective of CAD tools is to minimize the time for each iteration and to minimize the total number of iterations to reduce time-to-market.

Figure 4 illustrates classic and modern design paradigms. In the classic paradigm, design phases are sequential in nature with the underlying assumption that optimal implementation of each phase results in an optimal realization of the next. In the modern design paradigm to be supported by Nike, all design phases proceed in parallel. Within each design phase, infor-

mation is required from the other two. This necessitates ease of interaction between design tools at different design phases.

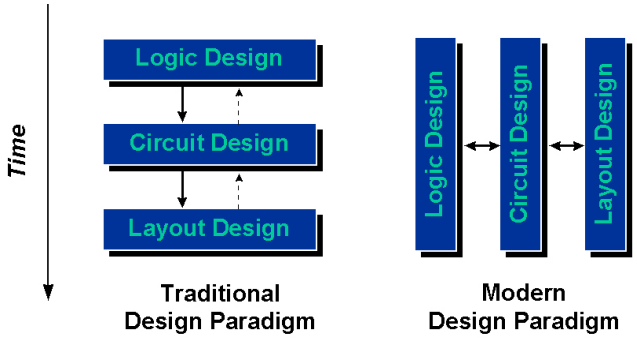

Figure 4: Design paradigms

Motivation

In terms of direct end-user benefit, data models are primarily driven by the productivity vector. This implies that data models should facilitate smoother automation/integration between different design tools with simpler work flows for certain tasks, and they should provide users with better capabilities to make tradeoffs/optimizations across design domains as well as between tools within the same domain.

These high-level requirements can be mapped into the following Nike system architecture goals:

- Enable a flexible configuration that facilitates a plug-and-play system architecture; different combinations of software components should be usable cooperatively. This is significant because system requirements will continue to evolve over the next several years.
- Achieve semantic consistency in data representation across layout, circuit, and logic domains. This will facilitates correct data transformations and exchange between tools in different domains.
- Promote reuse of common software components.
- Insulate individual CAD capabilities from persistent storage issues. This enables ease in changes to a storage mechanism as well as allowing multiple forms to exist transparently.

There is a trend in the EDA industry towards delivering a suite of inter-operable components rather than point tools. The next section presents a data model architecture that is helping us cope with this trend.

Data Model Architecture

As described above, the entire processor design cycle may be viewed as transformations between representations. At each representation, multiple CAD tools act as producers and consumers. A layout editor is an example of a design producer; simulators and design verification tools are examples of consumers. Data flow between producers and consumers typically transcends domain boundaries. For example, a layout routing tool may require information from a circuit timing engine; the timing engine in turn requires information from a layout parasitics calculator. Data modeling for the CAD domain is complicated by the diversity of design producers, consumers, and their singularities. This diversity also implies that different design tools need to view different aspects of a design. These considerations necessitate a modeling framework that can accommodate an assortment of data types and also be extensible to facilitate inclusion of new types. Our solution is to provide a framework that allows multiple levels of data models, related by specialization/generalization relationships, each tailored to the specific needs of applications. This facilitates interoperability and reuse of existing models. Semantic mappings between models promote ease of information
exchange. Figure 5 illustrates this modeling framework.

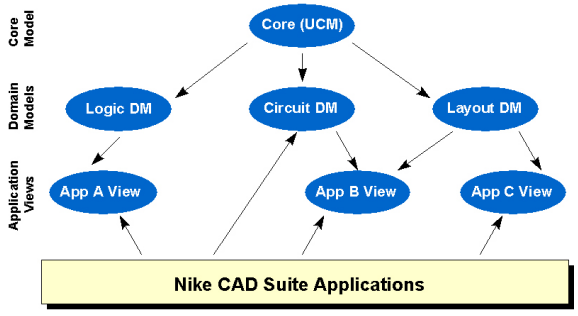

Figure 5: Data model framework

At the root of the framework is a unified core data model (UCM) that serves as the common vector between logic, circuit, and layout domains. It defines a

set of design entities that have consistent semantics across domains. These entities are primarily concerned with representing the hierarchy and connectivity[1] of a design, along with an interface to manipulate it. Domain models extend UCM with domain specific information. For example, the layout data model would add geometrical information. CAD applications can operate directly on UCM, a domain model, or an application view. An application view is either an extension of a data model with application-specific entities, or a suitably adapted perspective of the information in a data model. In all of the domain models and application views, UCM continues to be the common baseline. Having unified domain models allows the reuse of engines and minimizes I/O overhead. The reduction of the number of data models also facilitates the development of centralized cross-domain mapping services.

Figure 6 illustrates a usage scenario where UCM serves as a basis for enabling data-driven interactions between capabilities in the circuit and layout domains. Semantic mappings between data in the layout planner and circuit design environment are accomplished using a Mapper module. The Mapper has the ability to map from a design object in one domain to a corresponding design object in another domain; the mapping is performed at the granularity of UCM entities.

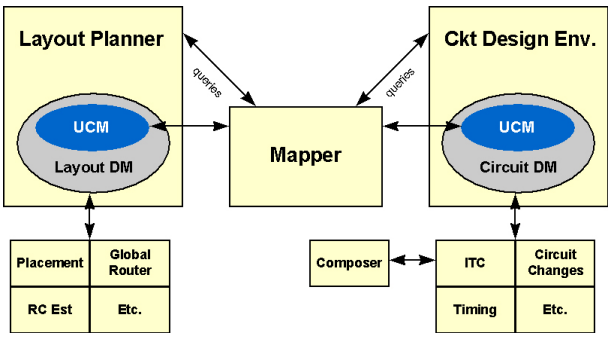

Figure 6: Sample usage scenario

Since the amount of time spent by each CAD tool on its task can be significant, it is also essential for interactions between tools to be incremental in nature. A designer must be able to operate on data in one domain and interactively see the effect of the change in another domain. As a concrete example, a designer can modify block placement in the layout planner and get interactive feedback on the effect of the new route on circuit timing. Although, data models alone cannot provide incremental interactions, they facilitate the development of modules and methodologies that can.

In summary, two key architectural goals have been addressed:

1. Facilitate interactive interactions across CAD tool boundaries.
2. Enable incremental iterations through the design cycle.

In the next section, we describe the software building blocks necessary to realize our vision of a modular architecture. We itemize the software tools and methodologies we implemented in Nike.

Software Infrastructure

Nike's architecture requires a strong focus on software quality. A common data model and a high level of module reuse introduce dependencies between projects that can magnify any software defects. Development at three sites adds additional complexity. To support high-quality development, Nike has defined and implemented a standard software development environment, including tools and methodologies, that are uniformly deployed to all Nike developers.

Standard Development Environment

The initial step was to itemize the software development tools being used across the department and set the Software Development Environment (SDE) Plan of Record (POR). The SDE POR lists all of the development tools and their versions that should be installed for each developer. This step is critical to ensure that all code libraries will be compatible and that common methodologies can be implemented.

Configuration Management

Our first priority was to choose a source code management tool for all sites and to define a common methodology for usage of the tool. A working group consisting of members from each site assessed best-in-class configuration management tools. After an evaluation and pilot usage, we agreed to purchase Rational's Clearcase *. Within several months, the tool and a common usage methodology were deployed to all the developers in the department.

The methodology defines everything from directory

[1] Since a chip is very complex, it is natural to decompose it into smaller components. Each component is successively decomposed into smaller, more manageable components, thus creating a *hierarchy* of components. The connections between these components are referred to as design *connectivity*.

* All other trademarks are the property of their respective owners.

structure for the code and libraries to tagging of versions and naming conventions for branches. The result is that all Nike developers are now using the identical setup so that developers from one group can easily navigate the source code from another group. In addition to the above features, when using the configuration management system's branch-and-merge capability, developers can create and maintain multiple variants of their software concurrently. This allows them to easily move from the latest nightly build to a previous release and vice-versa.

In order to facilitate a high level of code sharing across the sites, we implemented an additional component of the configuration management tool that enables cross-site sharing of code repositories, namely Rational's Multisite*. After a developer has revised code and checked it in at one site, these changes are synched up at the remote site at intervals of 15 minutes. The replication is transparent to the user. This means that developers can access all modules in real time, whether they are being developed locally or remotely. A developer in Haifa cannot tell the difference between a module developed in Haifa and one developed in Santa Clara. Developers can share modules on the spot and debug integration problems over the phone. They can iterate this process easily and efficiently, as if they are sitting in the same building. The capability of instant code sharing is key to the tight integration between our performance verification tools and physical design tools, and to enabling the module sharing and reuse detailed earlier.

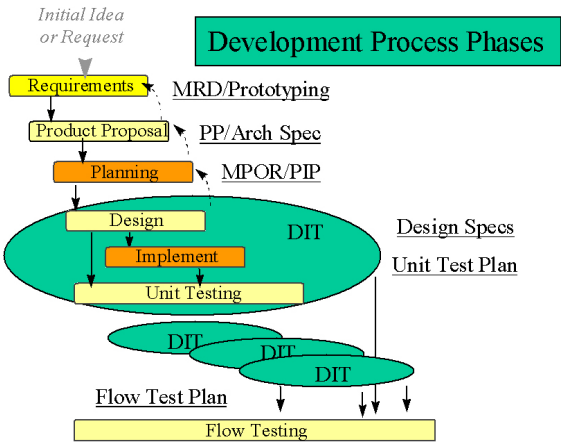

Figure 7: Software development cycle

Software Development Cycle

Another important component of the software infrastructure is the software development cycle. As shown in Figure 7, we defined an iterative software development process based on Design, Iterate, and Test (DIT) cycles. Each project breaks their development into small tasks whose DIT cycle is supposed to last no longer than 12 weeks. DIT cycles are designed to enable frequent synchronization points for validation. Once a quarter, we have a synchronization point where all the tools and modules synchronize to the same version of all libraries across all sites. This is critical in order to enable the modular architecture to work. The component sharing creates a large number of dependencies that we need to manage; the frequent synchronization points minimize the number of versions that need to be supported for each library.

As the common modules mature and stabilize, we have detailed a plan to move to weekly synchronization of the Nike libraries and eventually daily. Initially, there were too many frequent changes to the interfaces of the common modules to sync up more than once a quarter. In order to monitor the stability of the modules, we are tracking defects in the code through a bug-tracking system. After each synchronization point, bugs found in the libraries are reported internally and tracked in this system.

As part of the design phase, we require all Nike tools to write a Market Requirements Document, Product Proposal, External Product Spec, Internal Product Spec (when relevant) and a Test Plan. Each document in the design phase is approved by key customers, the system architects, and the software architects. Templates for all the documents are available for developers to help guide them through the process. In addition, we view prototype development as an important tool for gathering requirements and customer feedback and assessing new technologies. Prototyping may take place at any stage in the development cycle.

In order to achieve high quality in the code and design, we have implemented design reviews and code inspections across the organization. All critical code must be inspected, and authors are required to address major defects before an inspection is closed.

Validation

In the area of software testing, Nike is going for a breakthrough in both quality and productivity. After evaluating several test management systems from external vendors, we decided they did not meet our specific needs. Instead we chose to develop and deploy Olympus, an internally developed test management tool. Currently the first phase of development on Olympus is complete, and we are piloting the tool in each site. For improved productivity, Olympus pre-

sents the developer with a sophisticated front end through which he or she can complete test writing, regression building, golden results updating, and output analysis. Olympus integrates a user-friendly front end through which the developer creates and runs tests and regression on top of a back end that stores test data in an SQL database. Since all test data is centrally located in a database, multiple groups, across both sites, can share test data. Additionally, the details of the test are stored in the database and can be used for test planning even before the tests are completed. The database also adds an important dimension to our software development environment. Nike's testing is observable. Program management can query the database and find out how many tests have been created in the past month, how many have passed successfully, and how many are still not ready for regression.

Conclusion

In Nike, a major thrust of our development has been centered on defining a robust and modular architecture. The architecture is still evolving as development progresses. Our challenge is to sustain the architectural principles over the lifetime of the tool suite.

To date, we have invested significant effort in the definition phases of the data model and the infrastructure. We have observed that there is an overhead associated with software specification and definition due to the complexity of module sharing. However, we have also seen a significant reduction in the cost of implementation and integration. We perceive that this benefit will be even greater in the future once the foundation is complete and fully deployed.

Along with the benefits of a common data model, there are some associated risks. If the data model is not well managed, there is a potential for the data size to grow too large and complex. There is also a risk of organizational boundaries inhibiting development of shared modules if the sharing is not encouraged by management.

The changing design paradigm and constant evolution of technology brings forth a new set of challenges. A data model driven architecture takes us a step closer to a utopian "integrated, interactive, and incremental" system.

Acknowledgments

We thank the members of the Nike Product Engineering teams in Santa Clara and Haifa who have designed and implemented the software infrastructure as described here: Mark Ball, Shiri Cohen, Nina Galperovich, Miriam Kreisler, Ed Langlois, Neela Majumder, and John Ramirez. We also thank members of Nike Globals and Nike FCDE teams who were an integral part of defining and implementing the Nike data model architecture: Sridhar Boinapally, Vanco Burzevski, Yi-Hung Chee, Gil Kleinfeld, Zoya Korovin, Ronen Moldovan, Ran Ron, and Eric Tse. We thank the members of the Nike Architecture Board for playing a vital role in defining and consolidating the software architecture: Doug Braun, Yi-Hung Chee, Ganapathy Kumar, Mosur Mohan, and Siang-Chun The.

We thank the members of the Nike Product Engineering teams in Santa Clara and Haifa who have designed and implemented the software infrastructure as described here: Mark Ball, Shiri Cohen, Nina Galperovich, Miriam Kreisler, Ed Langlois, Neela Majumder, and John Ramirez. We also thank members of Nike Globals and Nike FCDE teams who were an integral part of defining and implementing the Nike data model architecture: Sridhar Boinapally, Vanco Burzevski, Yi-Hung Chee, Gil Kleinfeld, Zoya Korovin, Ronen Moldovan, Ran Ron, and Eric Tse. We thank the members of the Nike Architecture Board for playing a vital role in defining and consolidating the software architecture: Doug Braun, Yi-Hung Chee, Ganapathy Kumar, Mosur Mohan, and Siang-Chun The.

References

[1] Semiconductor Industry Association, *National Technology Roadmap for Semiconductors*, 1997.

Authors' Biographies

Veerapaneni Nagbhushan is currently a Nike software architect. Prior to that, he worked on several CAD tools at Intel. He holds a BE in electrical engineering from Birla Institute of Technology and Science and a M.S. in computer engineering from Syracuse University. Nagbhushan has been with Intel since 1987. His e-mail is vnagbhus@scdt.intel.com.

Yehuda Shiran is Nike Product Engineering Manager and Program Manager in Haifa. He holds a BSME and an MSME from the Technion in Haifa, a Ph.DME and an MSEE from Stanford, and an MBA from Haifa University. Yehuda has been with Intel since 1991. He previously worked in various Silicon Valley CAD development companies. His current interests include software development management infrastructure and software development discipline. His e-mail is yehuda.shiran@intel.com.

Satish Venkatesan is a senior CAD engineer in Santa Clara. He holds a doctorate in computer engineering from the University of Cincinnati and a BE in electrical engineering from the University of Roorkee. Satish has been with Intel since 1996. His e-mail is satish@scdt.intel.com

Tamar Yehoshua is currently managing the Nike Product Engineering team in Santa Clara and the PowerCAD team that provides CAD tools for low-power design. She holds a BA in applied mathematics from the University of Pennsylvania and an MS in computer science from the Hebrew University in Jerusalem. Tamar joined Intel's Design Technology group in 1993 where she has worked in CAD tool development and has held management roles. Prior to joining Intel, Tamar worked at the Institute for the Learning Sciences at Northwestern University. Her e-mail is tamar.yehoshua@intel.com.